Comp 311: Sample Midterm Examination

October 17, 2020

Synopsis of the Racket Language for this Exam

All programs should be written in the DrRacket Intermediate Student Language with lambda. This language includes the constructs: `define-struct`, `define`, `cond`, `if`, `and`, `or`, `local`, `lambda`, `error` and function application together the primitive types: number, boolean, symbol, string, `(listOf` T`)` for any type T, and n-ary function types $A_1 ... A_n -> B$ for any types A_1, ..., A_n, B. The primitive operations include

- all of the common arithmetic functions including `add1`, `sub1`, `+`, `-`, `*`, `-`;
- the standard relational operations `<`, `<=`, `=` `>=`, `>`;
- the boolean operation `not`
- the `equal?` operation on all data values;
- the list constructors `empty` and `cons` and accessor `first` and `rest`;
- recognizers for all of the major types and some commonly used subsets of those types including: `number?` `integer?`, `positive?`, `negative?`, `even?`, `odd?`, `rational?`, `inexact?`, `exact?`, `boolean?`, `symbol?`, `empty?`, `cons?`, `procedure?`;
- the `list` library functions `list`, `append`, `length`, and `reverse`; and
- the functional library functions `map` and `filter`.

The language constants include all numeric constants, all symbols, `true`, `false`, `empty`, and all list constant abbreviations of the form `'(a (b) c)`.

This description of the Intermediate Student Language with lambda should be sufficient to do all of the exam problems but it is not exhaustive. You may use language features and library functions included in the Intermediate Student Language with lambda except those that are explicity forbidden in the statement of a particular problem.

Note: This sample midterm is longer than what will actually be administered remotely as the Comp 311 midterm, but it covers the topics and varieties of questions that could potentially appear on the actual midterm more thoroughly than a *representative* sample midterm would. This sample midterm was created from an old exam that predates Racket and many of the courses in the existing Computer Science curriculum; there may be errors and omission in converting the text to use Racket and Comp 311 conventions. Please send email to the instructor if you notice any mistakes.

Enjoy!

Problem 1. (10 points) Given the Racket program:

```
(define (contains? los s)
  (cond [(empty? los) false]
        [else (cond [(equal? s (first los)) true]
                    [else (contains? (rest los) s)])]))

(define (& x y)        ; & is a legal function name in Racket
  (cond [x y]
        [else false]))

(define (unary-compose f g)
  (lambda (x) (f (g x))))
```

hand-evaluate each of the following **four** Racket expressions one-step-at-a-time. Use ellipsis where the context is clear. Try to fit each step on one line. *Omit repeating the definitions above.*

1. `(contains? (list 'a) 'b)`

2. `(& (= 1 0) (contains? empty 'c))`

Problem 1 cont.

3. `((unary-compose first rest) (list 'a 'b))`

4. `((unary-compose not (lambda (e) (contains? l 'a)) '(a b)))`

Problem 2. (10 pts) Which of the following expressions are Racket values? Answer **yes** or **no** for each expression.

- `foo`
- `'foo`
- `((lambda (x) (x x)) (lambda (x) (x x)))`
- `(list x y z)`
- `(lambda (x) (x x))`
- `cons?`
- `(list '+ 7 5)`
- `(foo '+ 7 5)`
- `(17)`
- `zero`

Problem 3. (10 pts) Write a definition for the function `mapcat` that given a function f returns the **concatenation** of the results of applying f to each element of the list. Note that `mapcat` is very similar to `map`, but `mapcat` has type

```
(alpha -> (listOf beta))  (listOf alpha)  ->  (listOf beta)
```

instead of

```
(alpha -> beta)  (listOf alpha)  ->  (listOf beta)
```

because it performs concatenation instead of `cons`ing. We are supplying the contract, purpose, and a minimal set of test cases for `mapcat`. You are responsible for writing the template instantiation and the code. You may not use the Racket library function `foldr` to write `mapcat`.

```
; mapcat: (alpha -> (listOf beta))  (listOf alpha)  ->  (listOf beta)
; Purpose  (mapcat f lob) returns the list produced by concatenating (f (first lob)),
;          ..., (f (last lob))

; Examples
(check-expect (mapcat (lambda (x) (list x x)) empty)  empty)
(check-expect (mapcat (lambda (x) (list x x)) (list 1))  (list 1 1))
(check-expect (mapcat (lambda (x) (list x x)) (list 1 2 3))  (list 1 1 2 2 3 3))

; Template Instantiation
#|
(define (mapcat f loa)





|#

; Code
(define (mapcat f loa)
```

Problem 4. (10 pts) Recall the `foldr` functional discussed in class (and introduced in the book in the exercises). It is defined by:

```
; foldr: (alpha beta -> beta) beta (listOf alpha) -> beta
; Purpose: (foldr op init loa) where loa = (list a1 a2 ... an) returns
;          (op a1 (op a2 ... (op an init) ...)).   If we use infix notation
;          for op, the result is: a1 op (a2 op ... (an op init) ...)

; Examples:
(check-expect (foldr + 0 empty) 0)
(check-expect (foldr + 0 (list 1)) 1)
(check-expect (foldr * 1 (list 1 2)) 2)
(check-expect (foldr + 0 (list 1 2 3)) 6)

; Code:
(define (foldr op init loa)
  (cond [(empty? loa) init]
        [else (op (first loa) (foldr op init (rest loa)))]))
```

Write a definition for `mapcat` (as defined in the previous problem) using `foldr` instead of explicit recursion. Since the preceding problem provided the contract, purpose, and examples for `mapcat` and the template instantiation for this definition of `mapcat` is degenerate, all that you have to write is the code.

```
(define (mapcat f loa)
```

Problem 5. (10 pts) Section 12.4 of the HTDP book shows how to write the function `arrangements` using a complex help function `insert-everywhere/in-all-words`. This definition of `arrangements` can be streamlined by using the library function `map` instead of structural recursion but still retains the help function `insert-everywhere/in-all-words`. It is possible to solve this problem with a simpler help function `insert-everywhere` provided that we use `mapcat` instead of `map`. Recall that the type of the function passed to `mapcat` is different than the type of function passed `map`.

The following code contains the contract, purpose, and minimal tests for the functions `arrangements` and `insert-everywhere` and the code for `insert-everywhere`. Write a definition for `arrangements` that uses `mapcat` and *only* the help function `insert-everywhere` plus core primitive functions on lists. Your definition of `arrangements` in conjunction with the definitions for `insert-everywhere` and `mapcat` should constitute a complete program for `arrangements`. Write both a template instantiation and the code.

```
;; Contract  arrangements: (listOf symbol) -> (listOf (listOf symbol))
;; Purpose:  (arrangements los) returns a list of all permutations of los

;; Examples
(check-expect (arrangements empty) (list empty))
(check-expect (arrangements '(a)) '((a)))
(check-expect (arrangements '(a b)) '((a b) (b a)))

; Template instantiation
#|
(define (arrangements los)




#|
;; Code
(define (arrangements los)







;; Contract  insert-everywhere: symbol (listOf symbol) -> (listOf (listOf symbol))
;; Purpose:  (insert-everywhere s los) returns a list of all lists of symbols
;;           obtainable from los by inserting s in some position within los.

;; Examples:
(check-expect (insert-everywhere 'a empty) '((a)))
(check-expect (insert-everywhere 'z '(b)) '((z b) (b z)))
(check-expect (insert-everywhere 'd '(b c)) '((d b c) (b d c) (b c d)))

(define (insert-everywhere s los)
  (cond [(empty? los) (list (list s))]
        [else (cons (cons s los)
                    (map (lambda (l) (cons (first los) l))
                         (insert-everywhere s (rest los))))]))
```

Problem 6. (10 pts.) Recall the definition of an `(binary-tree-map-of alpha)` (`(BTMof alpha)` from Homework 2. (In Homework 2, we wrote `(binary-tree-map-of alpha)` as `alpha-BTM` Given the structure definition,

```
(define-struct BTMNode (key val left right))
```

a `(binary-tree-map-of alpha)` (`(BTMof alpha)` for short) is either (*i*) the value `empty`; or (*ii*) `(make-BTMNode k s left right)` where `k` (the key) is a number, `s` is an *alpha*, and `left` and `right` are `(BTMof alpha)`'s. A `(binary-search-tree-map-of alpha)` (BSTMof *slpha*) is a `(BTMof alpha)` such that for every node `(make-BTMNode k s l r)` in `b`, every `key` in `l` is less than or equal to `k` and `k` is less than every `key` in `r`. In essence, the keys in a `(BSTMof alpha)` appear in sorted order (in the print-out of its value).

Write a definition for the function `BSTM-max` that takes a `(BSTMof alpha)` and finds the maximum *key* (extracted by applying `BTMNode-key` to a `BTMNode node in b`). On the degenerate `BSTM empty`, `BSTM-max` returns `#i-inf.0` which signfifies minus infinity, a number less than any exact number (integer or rational). Hence `(max n #i-inf.0) = n` for any exact number `n`. We are providing the contract, purpose, and minimal test data. Your task is to write the template instantiation and the code. Your program should run in time proportional the depth of the `b`— not in time proportional to the number of nodes in `b`. (Hint: you only recursive call should be `(BSTM-max (BTMNode-right b))` and you should only call it once.)

```
;; Node structure used to build BTMs and BSTMs
(define-struct BTMNode (key val left right))

; BSTM-max: (BSTMof alpha) -> number
; Purpose; (BSTM-max b) returns the maximum key in a BMTNode in b.
; Examples:
(define tree1 (make-BTMNode 50 'foo false false))
(define tree2 (make-BTMNode 20 'bar false tree1))
; (check-expect (BSTM-max empty) #i-inf.0)  ; check-expect rejects inexact inputs
(check-expect (BSTM-max tree1) 50)
(check-expect (BSTM-max tree2) 50)

; Template Instantiation
#|
(define (BSTM-max b)






|#
; Code
(define (BSTM-max b)
```

Problem 7. (10 pts.) Write a definition for the function `tree-map` that takes a function `f` of type `(number symbol` *alpha alpha* `->` *alpha*`)`, an intial value `init` of type *alpha*, and a `BTM` (which may or may not be a `BSTM` depending on its usage) and returns a value of type *alpha*. For the degenerate `BTM empty`, it returns the value `init`. For general `(BTMof` *alpha*`)`s, it returns the result of applying `f` at each node `(make-BTMNode k s left right)` to `k`, `s`, and the *alpha* values obtained recursively for `left` and `right`. We have provided the contract, purpose, and minimal test data for `tree-map`. Your task is to write the template instantiation and the code.

Note this description sounds more complex that it really is because of the type parameter *alpha*. To reduce confusion, think about the special case where *alpha* is `number`.

```
; tree-map: (number symbol alpha alpha -> alpha) alpha} BTM -> alpha
; Purpose (tree-map f init b) returns the result of "evaluating" the expression
;         generated by replacing every leaf in b by init and every node (make-bt k s l r)
;         by (f k s l' r') where l' and r' are the "translations" of l and r.

; Examples:
(define tree1 (make-bt 50 'foo empty empty))
(define tree2 (make-bt 20 'bar empty tree1))
(check-expect (tree-map (lambda (k val left right) (+ k left right)) 0 empty) 0)
(check-expect (tree-map (lambda (k val left right) (+ k left right)) 0 tree1) 50)
(check-expect (tree-map (lambda (k val left right) (+ k left right)) 0 tree2) 70)

; Template Instantiation:
#|
(define (tree-map f init b)




|#
; Code:
(define (tree-map f init b)
```

Problem 8. (10 pts)

- (5 pts) Write a new definition for `BSTM-max` that uses `tree-map` instead of explicit recursion. Note that the running time of this version of `BSTM-max` is proportional to the number of nodes in the tree, not its depth. (In fact, the code will work on `BTM`'s as well as `BSTM`'s.) You only need to write the code.

```
; BSTM-max: (BSTM alpha) -> number
; Purpose; (BSTM-max b) returns the maximum value of (BTMNode-key bn) for any BTMNode
;          in b.
; Examples:
(define tree1 (make-BTMNode 50 'foo false false))
(define tree2 (make-BTMNode 20 'bar false tree1))
; (check-expect (bst-max empty) #i-inf.0)  ; does not accept inexact arguments
(check-expect (BSTM-max tree1) 50)
(check-expect (BSTM-max tree2) 50)

; Code
(define (BSTM-max b)
```

- (5 pts) Write a definition for the function `BTM-depth` using `tree-map` that takes a `(BTMof alpha)` and returns its depth where depth is defined as the length of the longest path of `BTMNodes` from a leaf to the root in the tree. The depth of `empty` is 0.

```
; BTM-depth: (BTMof alpha) -> number
; Purpose; (BTM-depth b) returns the maximum number of BTMNodes on a path
;          from an empty leaf to the root.  The depth of empty is 0 since it does not conta
; Examples:
(check-expect (BTM-depth empty) 0)
(check-expect (BTM-depth tree1) 1)
(check-expect (BTM-depth tree2) 2)

; Code:
(define (BTM-depth b)
```

Problem 9. (20 pts.) The HTDP book suggests doing `merge-sort` from the bottom up. Their problem decomposition critically depends on two help functions (besides `merge`) whose direct implementations are not tail-recursive: `make-singles` (which we called `drop` in one of class lectures) and `merge-neighbors`. The code for the direct implementations of these two functions is given below together with contracts, purpose statements, and minimal tests. Your task is to rewrite both of these functions to use tail recursion and modify their minimal tests, if necessary, for consistency with behavior of the tail-recursive versions of these functions. Cross out any tests that you replace. Include contracts, purpose statements, minimal tests (akin to those given below), and template instantiations for any help functions you introduce. Note that the conversion to tail-recursive form typically reverses the order of list results because the tail-recursive form processes the elements of the input list in reverse order.

The function `merge-neighbors` appears on the next page. Note that the composition of these two functions does not yield a merge-sort function. The `merge-neighbors` function must be repeatedly applied in the body of merge-sort (reducing the length of an input list of length n to ceiling(n/2)) until it has length 1. Then the first (and only) element of this list is the sorted list is a sorted permutation of the input list given to merge-sort.

```
; make-singles: (listOf number) -> (listOf (listOf number))
; Purpose: (make-singles '(e1 ... en)) returns '((e1) ... (en))
; Structural Recursion Examples:
(check-expect (make-singles (list 1 2)) (list (list 1) (list 2)))
(check-expect (make-singles empty empty))
; Structural Recursion Code
(define (make-singles lon) (map list lon))    ; list is treated as a unary function
```

Problem 9 cont.

```
; merge-neighbors: (listOf (Ascending (listOf number))) -> (listOf (Ascending (listOf number)))
; Purpose: (merge-neighbors lol) returns a list of the merges of successive pairs of lists
;                                with an unmerged list at the end when (length lol) is odd
; Structural Recursion Examples
(check-expect (merge-neighbors empty) empty)
(check-expect (merge-neighbors (list (list 0))) (list (list 0)))
(check-expect (merge-neighbors (list (list 1) (list 0) (list 2)) (list (list 0 1)) (list 2))
; Code
(define (merge-neighbors lol) ; unusual struct recursion; recurs on (rest (rest lol))
  (cond [(empty? lol) empty]  ;; an empty lol should only be be passed by recursive calls.
        [else (let ([head (first lol)] [tail (rest lol)])
                (if (empty? tail) lol
                    (cons (merge head (first tail)) (merge-neighbors (rest tail)))))]))
```

Problem 10. (20 pts.) **Extra credit**.

Revise the `tree-map` function so that it can express a solution to `BSTM-max` that runs in time proportional to the depth of the tree. Show how to express the improved `BSTM-max` function using your revised `tree-map` function. Hint: the `BSTM-max` function that you wrote in Problem 8 above evaluates `(BSTM-max (left b))` when it is applied to a node `(make-BTMNode k s l r)`. If you make the function argument `f` to `tree-map` simulate call-by-name evaluation (which requires revising the definition of `tree-map`) of the arguments `l` and `r`, then `(tree-map f init b)` will descend into a subtree of `b` only when `f` (which must be re-written to "force" simulated call-by-name arguments) explicitly demands it.